REMARKS

The above amendment and these remarks are responsive to the communication from Examiner Majid Banankhah dated 25 April 2001, rejecting claims 1-8, all of the claims in the case.

35 U.S.C. 103

Claims 1-8 have been rejected under 35 U.S.C. 103 over Davidson et al. (U. S. Patent 5,630,136), in view of Periwal et al. (U. S. Patent 5,644,768).

Davidson and Periwal relate, respectively, to the OS primitive for locking such as mutex or sleep/wake. Typically, OS mutex primitives simultaneously wake all threads which are sleeping/waiting on a mutex-protected resources. It is then happenstance which thread actually makes it first to access the protected resource (such as by successfully obtaining Davidson's "baton"), and the others must go back to sleep/wait state. If, for example, 15 threads are waiting on a resource. As one becomes free, all 15 threads are awakened, one grabs the baton, and the other

14 all must return to wait state. This behavior causes thrashing on the multi-tasking OS's system "run" queue, where threads that are runnable, or waiting to be runnable, are tracked. By the time the last of the 15 threads is satisfied, it has make a lot of unsuccessful attempts to be serviced. A great deal of system resources is expended in managing such a run/wait operation.

On the other hand, applicant has provided a stationary queue. A "stationary queue" is not a queue in the traditional sense. As applicant explain in their specification:

> In accordance with this invention, the solution to this scarce resource management problem produces the same result as a queue or FIFO, but there is no memory actually associated with the queue or FIFO structure. No data or identifiers are moved into or out of this queue, hence the term 'stationary'. Only the read and write pointers of the queue need be maintained since the data that would ordinarily be stored in the queue is impliedly the counter, or pointer, values themselves. (Page 5, lines 2-10.)

Applicant's stationary queue, which includes two counters, insures not only fairness, but that only one thread at a time is awakened and given access, so system performance in resource constrained scenarios is maintained. There is no thrashing of a run/wait queue in the kernel.

As the Examiner correctly observes, a queue is a data structure. However, as is apparent to those of skill in the art, it is a very specific data structure, requiring storage for all the possible elements that the queue may hold, and a multitude of methods for manipulating the queue, including Init, Empty, IsEmpty, Enqueue, Dequeue. Applicant has provided a system and method which achieves queue-like behavior without any of the standard queue memory overhead. Applicant does not require memory for each element on the queue. In fact, storage requirements are the same regardless how many sleeping threads are to be placed in the stationary queue. Rather, applicant requires only two variables, or counters: the cumulative counter of threads forced to wait, and the cumulative counter of threads that have been served. These counters are used to generate thread-specific wake-up calls and by so doing avoid the thrashing that is inherent in Davidson and Periwal. It is as though applicant had a queue, but without the memory and methods required to instantiate a typical queue.

Periwal, on the other hand, teaches a mutex (record) and a nesting mechanism for preventing deadlocks when accessing the mutex. This is done to prevent deadlock conditions but, as with Davidson, there is no teaching of the fair, or FIFO, ordering of threads.

As the Examiner states, Davidson does fail to explicitly teach of a queue for allocating access to resources. In fact, Davidson and Periwal both fail to teach anything about enforcing order in his system. Periwal's reference to awakening "the next sleeping thread" (Col. 11, line 57) and to "subsequent threads or processes" (Col. 11, line 67) says nothing about enforcing order. No mention is made about how "next" or "subsequent" is determined, and in fact that order is undefined, as is apparent from an examination of the code in the Source Code Appendix of Periwald and by reference to on-line UNIX manual pages at the University of Virginia [http://www.cs.virginia.edu/cgi-bin/manpage?section=3T&topic=mutex], copy attached.

In the Source Code Appendix, Periwald provides for "awakening" at the following places for each of several operating systems:

Operating System	Col	Line	Description
Apollo	21	3	mutex_$unlock
OS/2	23	26	return DosReleaseMutexSem
Posix	25	49	return pthread_mutex_lock
Solaris	29	4	return mutex_unlock
VMS	31	26	return ISC_mustex_unlock
Windows	33	34	LeaveCriticalSection

Netware   37   16   return ISC_mutex_unlock

In each case, Periwald leaves the determination of which thread is to next have access to the mutex or critical section up to the operating system.

In the case of a mutex, as taught in the on-line UNIX manual page at the URL noted above, "By default, if multiple threads are waiting for a mutex, the order of acquisition is undefined." (Emphasis added). Periwald teaches no structure or method for assuring "fair order", or FIFO order, as variously specified in applicant's claims.

In the case of a critical section, as taught in the Microsoft Visual Studio Version 6.0 Help function on "EnterCriticalSection" (copied into a note dated 06/20/2001 09:40 AM from Bill Wilhelm to Shelley Beckstrand, copy attached) for mutual exclusion processing, each thread desiring to enter a critical section must first request and obtain ownership. When a thread having ownership, releases it, it is available for another thread. However, there is no structure or method taught by Periwald or provided by the Operating System for assuring that the "next" or "subsequent" thread to obtain ownership of the critical section is the next in fair, or FIFO, order. It is merely the "next" of all competing threads to win the race for ownership of the critical section.

Enforcing fairness, or order, is accomplished by applicant's invention. And yet, applicant does such without a queue, using instead a stationary queue (which he explains at Page 7 lines 14-15 includes a counter pair). A stationary queue is not a queue in the normal use of the term "queue", but rather a structure which presents queue-like behavior without being a queue. Applicant's invention achieves queue-like behavior (first-come, first-served) without the use of a real queue.

Each of applicant's claims distinguish Davidson and Periwald, as follows.

Claim 1, at lines 7-9, recites "a stationary queue for allocating access to said resource amongst said threads one-by-one in order of request"

Claim 2, at lines 9 and 10, specifies that "a next thread in line is to be re-animated and granted access <u>in FIFO order</u>."

Claim 3 depends from claim 2.

Claim 4 recites that a thread is awakened and granted access to a resource in fair order.

Claims 5-7 recites that requesting threads are animated one-by-one in order of request, or in fair order.

Claim 8, recites the components of a stationary queue comprising the counter pair mentioned above and, at lines 10 and 11, "a wakeup code routine for generating a unique run identifier when a next thread in line is to be re-animated and granted access to said resource" and "said wakeup code routine being responsive to said satisfied counter for generating said run identifier." This provides a "thread specific" wake up call, described in applicant's specification at page 8 lines 14-26 (which is copied in the appendix hereto for the purpose of making a spelling correction) using the two counters that applicant provides to wake up only the next thread. The inherent result is no contention or thrashing for ownership of the resource. This is not taught by Davidson or Periwald.

SUMMARY AND CONCLUSION

Applicant urges that the above amendments be entered, in the case of the specification and claim 1 as relating to minor spelling corrections not affecting the scope of the claims, and with respect to claim 2 as merely adding the FIFO limitation already in other claims, and the case passed to issue with claims 1-8. Attached is a version showing changes made to claims 1 and 2, and to the specification.

If, in the opinion of the Examiner, a telephone conversation with applicant's attorney could possibly facilitate prosecution of the case, he may be reached at the number noted below.

Sincerely,

G. W.Wilhelm, Jr.

By

Shelley M Beckstrand

Shelley M Beckstrand
Reg. No. 24,886

Date: 20 June 2001

Shelley M Beckstrand, P.C.
Attorney at Law
314 Main Street
Owego, NY 13827

Phone: (607) 687-9913
Fax: (607) 687-7848

a:amend2.wpd

VERSION WITH MARKINGS TO SHOW CHANGES MADE

In the specification

The paragraph beginning at page 8 line 14 has been amended to correct a spelling error.

-- If, in step 144, it is determined that the requested resource is available, in step 145 the thread uses the resource and, when finished, in step 152 returns the resource. In step 154 it is determined whether or not NFW counter 112 equals NSC [cunter] <u>counter</u> 110. If they are equal, there are no threads waiting on the resource, and processing returns to step 142 to await a request from a thread for the resource. If they are not equal, there is a waiting thread. In step 156, run ID 132 is created using the value of NSC 110 counter, which will then be incremented in step 158. In step 160, the thread identified by run ID 132 is awakened and made ready to run by the operating system. In step 162, that thread awakens, and in step 145, uses the resource.--

In the Claims

Claim 1 has been amended to correct a spelling error, and claim 2 has also been amended.

--1. [Twice Amended] A multi-tasking operating system for managing simultaneous access to scarce or serially re-usable resources by multiple process threads, comprising:

at least one resource;

a plurality of threads requesting access to said resource; and

a stationary queue for allocating access to said resource [amonst] amongst said threads one-by-one in order of request.--

2. [Amended] A multi-tasking operating system stationary queue for managing simultaneous access to scarce or serially re-usable resources by multiple process threads, the stationary queue comprising:

a sleep code routine for generating a unique block identifier when a process thread temporarily cannot gain access to said resource and must be suspended; and a wakeup code routine for generating a unique run identifier when a next thread in line is to be re-animated and granted access to said resource <u>in FIFO order</u>.